Cours 4
Appels système

Qu'est-ce qu'un appel système ?

Définition : un fichier de bibliothèque est une archive contenant un ensemble de fichiers objet qui peuvent être extraits de l'archive pour être liés à un programme.
Donc le code d'une fonction de bibliothèque est dans l'exécutable qui fait appel à cette fonction. Un utilisateur peut toujours ajouter des fonctions de bibliothèque.
Définition : un appel système est un point d'entrée dans le code du noyau.
Autrement dit, le code de l'appel système n'est pas dans l'exécutable qui fait cet appel, et aucun utilisateur ne peut ajouter d'appel système.
Quelques propriétés :

- Le code du noyau est très sensible. Tout usage incorrect peut détériorer le matériel ou planter le système (donc gâcher beaucoup de temps de travail, perdre des données). Ce code est donc protégé :
  - On ne peut y accéder autrement que par un appel système (on ne peut faire par ex un call direct à l'adresse d'une routine système)
  - Les paramètres de l'appel système sont contrôlés avant que la fonction ne soit exécutée
- Les appels systèmes disponibles ne sont pas les mêmes suivant le système (Windows, Unix, VMS, etc.) et, pour un même système, suivant l'architecture (ex : Unix sur PC, sur Sun, sur HP…)
- Chaque langage spécifie les appels systèmes qu'il utilise. En théorie, peu importe qu'ils varient d'une implémentation à l'autre, s'ils rendent les mêmes services. En fait
  - Certains services ne sont pas disponibles sur certaines plates-formes. Certaines spécifications ne peuvent donc pas être implémentées (ex : fork() sous Dos)
  - Les normes sont des normes industrielles. L'importance des enjeux explique qu'il y en ait plusieurs divergentes (ex.: BSD, Système V, OpenLook, Posix,…)
  - Le matériel et le logiciel évoluent sans cesse. La conception de certains appels système ne peut simplement pas encore être stabilisée (ex.: les procédures de communication entre machines à travers un réseau)..

Comment utiliser les appels système

La première question à se poser est : dans quel mesure le code écrit doit-il être portable ? Et combien de temps sera-t-il utilisé ?
Si le code risque de devoir un jour être porté sur d'autres machines ou d'autres systèmes, il faut absolument repérer les appels non portables et les isoler dans quelques fonctions qui pourront être testées et réécrites.
Ex: lecture des informations sur un fichier. L'appel système est :

```
#include <stat.h>
int stat(const char *path, struct stat *statbuf);
```

Quand on regarde dans stat.h sous Linux ou sous Windows, on trouve les définitions suivantes (simple souligné = le champs existe sous Windows, mais n'a pas la même signification; double souligné = il n'existe pas sous Windows.) :

```
struct stat                          /* gnu C sous Linux */
        {
  dev_t         st_dev;      /* le périphérique où le fichier est stocké*/
  ino_t         st_ino;      /* numero de fichier sur ce périphérique */
  mode_t        st_mode;     /* droits d'accès */
  nlink_t       st_nlink;    /* nombre de liens physiques */
  uid_t         st_uid;      /* numéro d'utilisateur (identifiant) du propriétaire */
  gid_t         st_gid;      /* numéro de groupe (identifiant) du propriétaire */
  dev_t         st_rdev;     /* type de périphérique (si fichier spécial)*/
  off_t         st_size;     /* taille, en octets */
  unsigned long st_blksize;  /* taille de bloc pour les E/S du système de fichiers */
```

```
  unsigned long st_blocks;    /* nombre de blocs alloués */
  time_t        st_atime;     /* date du dernier accès */
  time_t        st_mtime;     /* date de la dernière modification */
  time_t        st_ctime;     /* date du dernier changement (de nom, de droit,..) */
};

struct stat {                  /* Borland C, Dos ou Window 95 */
  short  st_dev,   st_ino;          /* st_ino sans signification */
  short  st_mode,  st_nlink;        /* st_nlink == 1 */
  int    st_uid,   st_gid;          /* sans signification */
  short  st_rdev;                   /* identique à st_dev */
  long   st_size,  st_atime;
  long   st_mtime, st_ctime;        /* identiques à st_atime */
};
```

Conclusion : tout ce qui concerne les propriétaires, l'accès aux périphériques, les dates de modification doit pouvoir être retrouvé dans le code.

Les principaux types d'appel système.

Sous Unix, le manuel donne environ 200 appels système, dont certains sont périmés et d'autres non implémentés. Il faut donc apprendre à lire rapidement une description. Voici quelques exemples des principaux types sous Unix (les appels qui existent sous le même nom sous Windows selon Borland C sont en italiques. Ils sont soulignés quand le contenu présente une grande différence). La commande **`man 2 nom_de_l'appel`** donne une description des appels système.

Administration :

`getuid(), getgid(),` identification d'utilisateur, de groupe
`gethostname(), sethostname(), uname(),` nom de la machine, système
`getrlimit(), acct()` ressources autorisées, enregistrement de l'activité des utilisateurs

Temps :

`alarm(), setitimer(), gettitimer(), stime(), time()` temporisation
`gettimeofday(), settimeofday()` date

Réseau :

`socket(), bind(), connect(), send(),` communication entre machines
`listen(), select(), socketpair()`
`ioctl()` paramétrage des entrées-sorties (locales ou réseau)
`getdomainname(), setdomainname()` administration du réseau

système de fichier (cf. cours 7):

`open(), close(), create()`[1]`, lseek(), read(), write(), truncate(),`
`link(), rename(), unlink(),`
`chmod(), chown(), chroot(), umask(), stat()` (fichier), `statfs()` (disques),
`chdir(), mkdir(), mknod(),`
`flock(), fsync(), sync(), mount(), umount()`

processus :

`fork(),execve(), system(),sleep(), pause(),exit()` création, mise en attente, terminaison
`mmap(), mprotect()` copie d'une partie de fichier en mémoire, partage de mémoire
`nice(),` priorité
`ptrace()` pour debuguer
`shutdown()` terminaison de tous les processus

[1] creat en Dos…